Stateless: an Optic Algebra Library

Jesús López-González
j.lopezgo@alumnos.urjc.es

Juan Manuel Serrano
juanmanuel.serrano@urjc.es

June 8, 2017

Abstract

Abstract goes here!

1 Introduction

Software industry as we know it lacks well-founded frameworks to program reactive systems in a modular way. This kind of systems hide an inner *state* to their environment, showing therefore a black-box behaviour. Take a university information system as a guiding example. Universities are structured in departments, and each has a budget. Even this simple definition does contain three kinds of entities: universities, departments and budget. Ideally, programming the evolving state of the whole system should be achieved by programming the state of each entity or subsystem, and finally composing them all.

In addition, composition should contemplate heterogeneity: departments could be stored in a local database while budget could be deployed as external microservices. Ultimately, any configuration should be supported, but it is essential for the system description to remain pure and decoupled from low-level details, irrelevant to the domain. Needless to say that it should never hamper optimal performance. Programming a declarative and modular system in this way is still a great challenge, but functional programming seems like a good starting point [Hughes, 1989].

There are plenty of technologies to develop reactive systems, but they are postulated as solutions to very specific problems, and consequently, difficult to be reused. Turning to the university example, we could have decided to program department evolution on a relational database, like *MySQL*. However, if large volumes of data arose in time, migrating to a distributed database such as *Redis* might become necessary. Given that situation, we would find it extremely difficult to adapt existing encodings to the new setting.

Fortunately, industry has detected this issue and new abstract tools have emerged to fill the gap. In this sense, we could have avoided the aforementioned problem by programming department evolution with *Slick*, a Functional-Relational Mapper that provides an *embedded domain specific language* [Hudak, 1996] which can be later interpreted into different databases. Similarly, we could find other situations where a common language could unify different state evolutions. Take *optics* as an example. They could be seen as a language to achieve complex transformations over immutable in-memory data. In this scenario, *Traversal* [O'Connor, 2011] could be used as the glue to keep university and department related.

We claim that many of these languages to access state could be further factorized, by providing a new language to unify them, while retaining reasonable levels of performance for a large class of applications. Moreover, we claim that optic concepts are ideal to abstract technology-specific details away. However, prior to that, they have to be taken to a more general setting: *optic algebras*.

An optic algebra represents the essence of the state transformation associated to that optic. Generally speaking, optics were conceived to tackle a very specific problem, achieving complex transformations over immutable in-memory data structures. For instance, a *lens* [Foster et al., 2005] knows how to transform certain data structure —the focus— which is contextualized in a larger one —the whole. Regarding our university, this abstraction could perfectly describe the existing relationship between a department and its associated budget. Having said so, what do we mean by the essence of a lens?

The essence of a lens requires removing all those aspects which are specific to in-memory transformations. If we take into account the classic definition of a lens in Scala [Morris, 2012]:

```scala
case class Lens[S, A](
  get: S => A,
  set: A => S => S)
```

one could easily identify several accesses to the in-memory state *S*. If we abstract away from those aspects, ie. if we conceal the state, we end up with the optic algebra associated to lens:

```scala
case class LensAlg[F[_], A](
  get: F[A],
  set: A => F[Unit])
```

This algebra —which bears a strong resemblance to *MonadState* [Gibbons, 2012]— determines the existence of a focus that could be either read or updated, but it keeps itself unaware of the details underneath *F*. Similar derivations could be carried out over other optics, resulting in new optic algebras.

Composing optic algebras —as they were introduced in the previous paragraph— is not straightforward at all. In fact, the situation is very similar to the one that emerges when trying to compose optics in their classic representation. In this sense, Van Laarhoven and recently Profunctor optics [Pickering et al., 2017] have been postulated as alternative representations to improve composition. Taking them as inspiration, we created a new representation for optic algebras. Since it heavily relies on natural transformations [Pierce, 1991], we refer to it as *natural* representation. This representation not only improves the composition up, but also introduces the notion of heterogeneity, as we will see in further sections.

Optic algebra instances for the most widespread deployments should be provided. Given this scenario, by defining our system in terms of optic algebras, we should be able to get our logic implemented once and for all. Later, we could find it interpreted into a method call, a database access, a microservice invocation, or even as a combination of many of them.

We claim that optic algebras in their natural representation lead to modular software designs. For instance, a traversal algebra could establish the connection between the university and each department. On the other hand, a lens algebra could describe the relationship between departments and budget. Then, we could compose both algebras to conform the whole system. Additionally, optic algebras will enable us to do it in an heterogeneous way, with many free instances ready to go, offering a reasonable performance in the majority of cases.

2 Background

3 Deriving Optic Algebras

4 Natural Composition and Heterogeneity

5 Instances for Free

6 Conclusions

7 Acknowledgements

References

[Foster et al., 2005] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2005). Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246.

[Gibbons, 2012] Gibbons, J. (2012). Unifying theories of programming with monads. In *International Symposium on Unifying Theories of Programming*, pages 23–67. Springer.

[Hudak, 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196.

[Hughes, 1989] Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

[Morris, 2012] Morris, T. (2012). Asymmetric lenses in scala.

[O'Connor, 2011] O'Connor, R. (2011). Functor is to lens as applicative is to biplate: Introducing multiplate. *arXiv preprint arXiv:1103.2841*.

[Pickering et al., 2017] Pickering, M., Gibbons, J., and Wu, N. (2017). Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857*.

[Pierce, 1991] Pierce, B. C. (1991). *Basic category theory for computer scientists*. MIT press.